Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-245

Users Guide for SnadiOpt: A Package Adding Automatic Differentiation to Snopt*

by

E. Michael Gertz
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439

Philip E. Gill and Julia Muetherig
Department of Mathematics
University of California, San Diego
La Jolla, California 92093-0112

Mathematics and Computer Science Division

Technical Memorandum No. 245

January 2001

gertz@mcs.anl.gov
http://www.mcs.anl.gov/~gertz/

pgill@ucsd.edu
http://www.scicomp.ucsd.edu/~peg/

jmueth@ucsd.edu
http://www.scicomp.ucsd.edu/~julia/

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by National Science Foundation Grant CCR-95-27151.

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States Government and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor The University of Chicago, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or The University of Chicago.

Contents

Users Guide for SnadiOpt: A Package Adding Automatic Differentiation to Snopt

by

E. Michael Gertz, Philip E. Gill, and Julia Muetherig

Abstract

SnadiOpt is a package that supports the use of the automatic differentiation package ADIFOR with the optimization package Snopt.

Snopt is a general-purpose system for solving optimization problems with many variables and constraints. It minimizes a linear or nonlinear function subject to bounds on the variables and sparse linear or nonlinear constraints. It is suitable for large-scale linear and quadratic programming and for linearly constrained optimization, as well as for general nonlinear programs.

The method used by Snopt requires the first derivatives of the objective and constraint functions to be available. The SnadiOpt package allows users to avoid the time-consuming and error-prone process of evaluating and coding these derivatives. Given Fortran code for evaluating only the *values* of the objective and constraints, SnadiOpt automatically generates the code for evaluating the derivatives and builds the relevant Snopt input files and sparse data structures.

Keywords: Large-scale nonlinear programming, constrained optimization, SQP methods, automatic differentiation, Fortran software.

1. Introduction

This is the users guide for SnadiOpt, a package that adds the automatic differentiation capability to the nonlinear optimization package Snopt [5]. SnadiOpt uses the source-to-source automatic differentiation package ADIFOR to perform the differentiation.

1.1. Problem Types

Snopt is a collection of Fortran 77 subroutines for solving a *nonlinear programming problem* assumed to be stated in the following form:

NP	$\min\limits_{x}$ (or max)	$F_{obj}(x)$	
	subject to	$l \leq x \leq u,$	$L \leq F(x) \leq U,$

where u, U, l, and L are constant vectors of upper and lower bounds, $F(x)$ is a vector of smooth linear and nonlinear problem functions, and $F_{obj}(x)$ denotes the component of F to be minimized or maximized.

Note that upper and lower bounds are specified for all variables and constraints. This form allows full generality in specifying various types of constraint. Special values are used to indicate absent bounds ($l_j = -\infty$ or $u_j = +\infty$ for appropriate j). Free variables and free constraints ("free rows") are ones that have both bounds infinite. Similarly, fixed variables have $l_j = u_j$, and equality constraints have $L_j = U_j$

The method used by Snopt requires that the elements $J_{ij}(x) = \partial F_i(x)/\partial x_j$ of the Jacobian matrix of first derivatives be known at any point x. In practice it is often inconvenient or impossible to code the derivatives, and so Snopt allows the user to code as many derivatives as is convenient. Snopt then estimates unknown derivatives by finite differences, by making a call to F for each variable x_j whose partial derivatives need to be estimated. However, finite differences reduce the reliability of the optimization algorithm and can be expensive if there are many such variables x_j. The SnadiOpt package allows the user to avoid the time-consuming and error-prone process of evaluating and coding derivatives without the need for Snopt to compute finite differences.

Often, an element J_{ij} is constant, which implies that variable x_j occurs only linearly in the problem function $F_j(x)$. If a significant number of these constant elements are zero, then J is known as a sparse matrix, and Snopt uses a sparse matrix format to store only the nonzero elements of J. SnadiOpt automatically identifies constant and zero Jacobian elements by using a scheme that evaluates the Jacobian at a number of points close to the starting point (see Section 2). Given Fortran code for evaluating only $F(x)$, SnadiOpt automatically generates code for evaluating J and builds the relevant Snopt input files and sparse data structures.

1.2. Why Automatic Differentiation?

Writing code for the derivatives of $F(x)$ is difficult, time consuming, and error prone, especially when problems involve many variables and constraints. Automatic differentiation (AD) tools, in this case ADIFOR [1, 2], quickly provide correct and numerically accurate derivative functions from the code used to evaluate the objective and constraint functions.

Prior to the wide availability of AD software and AD-based modeling languages, numerical differentiation was the only alternative to providing derivative code. Unfortunately, numerical differentiation is an inherently unstable process that causes both theoretical and practical difficulties for nonlinear solvers. Numerical differentiation places a severe theoretical limit on the accuracy of the solution that may be computed by an algorithm (see, e.g., [7, Chapter 8]). In practice, code that uses numerical differentiation tends to need

more iterations to find a solution than does code that uses exact derivatives. Furthermore, code that uses numerical differentiation is typically less robust and will fail to find solutions for problems that might have been solved if analytic derivatives were supplied. Notwithstanding these difficulties, numerical differentiation was often used to avoid the high cost of hand-coding the exact derivatives.

With modern AD tools, derivative code may be quickly obtained, leading to a significant increase in user productivity—even on simple problems. Moreover, functions can now be differentiated that were once considered too complex to be coded by hand. An example is a function defined in terms of the output from an ordinary differential equation solver. ADIFOR has been successfully applied to such functions.

Automatic differentiation allows users to develop models quickly. This increase in productivity makes optimization software a much more useful tool for scientists, who often wish to experiment with different objective functions and different sets of constraints.

1.3. ADIFOR

ADIFOR is a robust, mature automatic differentiation tool developed through a collaborative project between the Mathematics and Computer Science Division at Argonne National Laboratory and the Center for Research on Parallel Computation at Rice University. The package is a source-to-source translator for functions written in Fortran 77. It is widely available and runs on many popular platforms. Moreover, the source code for the ADIFOR libraries (but not the translator) is provided, making it possible to compile and run ADIFOR-generated code on most platforms.

ADIFOR implements the forward mode for automatic differentiation (although it does make some use of the reverse mode internally). As a practical matter, this means that ADIFOR-generated code will tend to run most quickly if the number of variables is not much larger than the number of constraints.

1.4. Who Should Use This Package

It is usually wise to try automatic differentiation before attempting to code derivatives by hand. Some users may be more comfortable using modeling languages with an automatic differentiation capability (see, e.g., AMPL [4] and GAMS [3]). SnadiOpt is intended for those who prefer to code in Fortran, or need to make use of existing Fortran software. Such users should find that this package can provide derivative code quickly and efficiently.

Programmers who typically write in C or Fortran 90 might like to consider developing their models in Fortran 77 so that they may use this package to obtain derivatives. A C programmer should be able to learn enough Fortran 77 to formulate simple to moderately complicated models within a few hours. C has many features that Fortran 77 does not, but it is exactly those features that can make the automatic differentiation of C code problematic.

1.5. How to Read This Manual

Section 1.6 summarizes the four steps needed to define and solve an optimization model using SnadiOpt and Snopt. Section 2 describes the main features of the SnadiOpt package and provides some background on the mechanics of automatic differentiation (this section may be omitted on first reading). Section 3 describes the subroutines that must be provided by users to define their model. Section 4 descibes the invocation of the Perl script `snadiopt.pl` that automatically generates all input files for ADIFOR. This discussion includes detailed information on the various files generated by SnadiOpt (see Section 4.3). Finally, Section 5 discusses the use of the Make utility to automatically differentiate the problem files and build an executable file ready for execution.

1.6. Basic Usage

In the simplest case, the SnadiOpt package can be used to solve problem NP in four steps.

Step 1. Construct a file `prob.f` containing Fortran subroutines `usrini` and `usrfun` (see Section 3). Subroutine `usrini` initializes all data associated with the model, including the bounds l, u, L, and U, and the component of F that defines the objective function. Subroutine `usrfun` defines the values of the problem functions F at a given value of x. A Fortran main program is not required. However, subroutine `usrini` must define the lengths of all arrays used to define the model.

Step 2. Invoke the Perl script `snadiopt.pl` to generate the input files needed by the automatic differentiation package ADIFOR. The syntax of the call is

```
% snadiopt.pl -o prob prob.f
```

(The symbol "`%`" is the shell prompt and is not to be typed). This generates a number of auxiliary files with the prefix `prob_` (see Section 4).

Step 3. Build the executable file for the model `prob`. This step uses ADIFOR to generate the differentiated subroutines needed for Snopt, compiles them and links them with the Snopt libraries. All these tasks are performed by using the Make utility, where the `makefile` is one of the files automatically generated by `snadiopt.pl` (see Section 5). To start the build process, type

```
% make prob
```

Step 4. Solve the optimization problem by typing

```
% ./prob
```

Any output from the run will be written to the files defined in the subroutine `usrini`.

1.7. Additional Resources

All users should read the Snopt users guide [6], which details the many user options available in Snopt that may be set by providing a "specs" file. The users guide describes the algorithm and its output and answers many questions about the performance of the optimizer on a particular model.

SnadiOpt tries to insulate the user from the details of invoking ADIFOR, but users may wish to read the ADIFOR users guide. In particular, while ADIFOR is very robust, it is possible to write Fortran code that fools ADIFOR, and the manual will explain how to avoid this pitfall.

For general information on optimization, we recommend that users explore the NEOS guide on the Web: `http://www.mcs.anl.gov/otc/Guide/index.html`.

More information about automatic differentiation in general, and ADIFOR in particular, may be found on the Argonne National Laboratory automatic differentiation Web page: `http://www.mcs.anl.gov/autodiff`.

Users with complicated functions for which the automatically differentiated code appears to be unacceptably slow can often accelerate their code by refactoring it. Several technical reports on the automatic differentiation page describe how to do this.

The authors of this package also maintain Web pages. Philip Gill's page[†] has links to published papers and technical reports on Snopt. Michael Gertz maintains a Web page[‡] at Argonne National Laboratory.

[†] `http://scicomp.ucsd.edu/~peg/`
[‡] `http://www.mcs.anl.gov/~gertz/`

2. Automatic Differentiation

Automatic differentiation is the process of producing code that evaluates the derivatives of a function from code that evaluates the function itself. It is closely related to symbolic differentiation but differs from it in important ways. Symbolic differentiation takes the mathematical expression for a function and produces another expression that represents the derivative of that function. Unlike symbolic differentiation packages, automatic differentiation packages:

- understand programming concepts such as loops, branches and subroutines and
- use intermediate quantities and the chain rule to avoid potentially exponential growth in the size of the resulting code.

ADIFOR is a source-to-source translator: it takes as its input a function expressed as a Fortran 77 subroutine and generates Fortran code that computes the derivatives of the dependent variables with respect to the independent variables. Suppose $F : \mathbb{R}^n \to \mathbb{R}^m$ is a function and

```
subroutine func  ( x, n, F, m )
integer            m, n
double precision   x(n), F(m)
```

is a Fortran subroutine that will compute the value of F at any given x. Let $J(x) = F'(x)$ be the Jacobian of F. ADIFOR will produce code that computes $J(x)S$ for any $n \times p$ matrix S with $p \leq n$. Thus, if $S = I$, ADIFOR will compute the Jacobian itself.

In many optimization models, certain terms that occur in F will be linear and will result in constant elements in $J(x)$. SnadiOpt does not require that these elements occur in a particular part of $J(x)$, but for the sake of discussion, let us assume that $J(x)$ has the following structure,

$$J(x) = \begin{pmatrix} N_{11} & L_{12} \\ L_{21} & L_{22} \end{pmatrix},$$

where the elements of L_{12}, L_{21}, and L_{22} are constant and the elements of N_{11} may or may not be constant, but all rows and columns of N_{11} contain at least one nonconstant element. Any Jacobian may be transformed to a matrix with this structure by permuting the constraints (rows) and variables (columns). Snopt is designed to exploit the constant elements in J. For instance, the constraints corresponding to $\begin{pmatrix} L_{21} & L_{22} \end{pmatrix}$ are linear, and Snopt will maintain feasibility with respect to the linear constraints. Because the ADIFOR generated code computes $J(x)S$, SnadiOpt is able to choose an S in a manner that avoids the need to reassign L_{12}, L_{21}, and L_{22} every time J is required.

Snopt is designed to solve problems with sparse derivatives. These are problems for which many of the elements of $J(x)$ are identically zero. SnadiOpt determines the sparsity pattern for the Jacobian and identifies the constant elements automatically. To make this determination, SnadiOpt computes the value of $J(x)$ at two random perturbations of a user-supplied initial point x_0. If an element of the Jacobian is the same at both points, then it is taken to be constant. If it is zero at both points, it is taken to be identically zero. The random points are not chosen close together, so the heuristic will correctly classify the Jacobian elements in the vast majority of cases. Snopt validates the computed derivatives, linearity pattern, and sparsity pattern at the point x_0, by comparing the supplied values to values computed using numerical differentiation. This additional test at a third point makes it unlikely that an incorrect sparsity or linearity pattern will be used.

Of course, it is possible to fool this heuristic. SnadiOpt cannot deal well with functions for which the sparsity pattern or linearity pattern in a (relatively large) region around x is

not representative of the sparsity or linearity pattern of the function as a whole. Computing a sparsity pattern for such a function would require significant additional user intervention. Because we are uncertain of the demand to minimize such functions, we have opted for the simpler user interface. We welcome examples of real-word optimization models that fall into this category.

Once SnadiOpt has computed the sparsity and linearity pattern and the appropriate S to minimize recomputation of the derivatives of linear elements, it calls Snopt as a "black-box" optimization routine. This means that it presents the optimization data to Snopt in the same format as a hand-written routine for computing the derivatives. Users have full access to all the options and features of Snopt and can link the resulting code with their own code (subject, of course, to any licensing restrictions.)

3. User-Supplied Subroutines

In order to use SnadiOpt, the user must provide the following Fortran routines:

`usrfun` (§3.1) Defines the functions $F_i(x)$.
`usrini` (§3.2) Defines the actual dimension of the problem and initializes all data needed by Snopt. The workspace for SnadiOpt is also assigned here.

The user routines `usrfun` and `usrini` have fixed parameter lists but may have any convenient name. The names of the parameters may also be chosen by the user.

3.1. The Function Definition Routine

The user must provide a subroutine that calculates the vector F of objective and constraint functions at a given point x.

```
      subroutine usrfun( Status, mode, x, n, F, neF,
     &                   cu, lencu, iu, leniu, ru, lenru )
      integer            Status, mode, neF, n
      double precision   F(neF), x(n)
      integer            lencu, leniu, lenru
      character*8        cu(lencu)
      integer            iu(leniu)
      double precision   ru(lenru)
```

On entry:

`Status` indicates the first and last call to `usrfun`.

If `Status` $= 0$, there is nothing special about the current call to `usrfun`.

If `Status` $= 1$, Snopt is calling the subroutine for the first time. Some data may need to be input or computed and saved.

If `Status` ≥ 2, Snopt is calling the subroutine for the last time. The user may wish to perform some additional computation using the final solution.

If the nonlinear functions are expensive to evaluate, it may be desirable to do nothing on the last call, by including a statement of the form

```
if (Status .ge. 2) return
```

at the start of the subroutine.

`x(n)` contains the point at which the problem functions are to be evaluated.

`cu(lencu), iu(leniu), ru(lenru)` are character, integer and real scratch arrays. These arrays may be used to store any information that needs to be saved between calls to `usrfun`.

On exit:

`F(neF)` holds the values of the objective and constraint functions computed at `x`. The objective is component `F(ObjRow)`, as defined in `usrini`.

`mode` is used to communicate between Snopt and the user. If the user is unwilling or unable to evaluate the function at the current point then `mode` should be set to -1. Snopt will try to provide an alternative point at which to evaluate the function.

If for some reason the user wishes to terminate solution of the current problem, `mode` should be set to a negative value (other than -1).

`cu(lencu), iu(leniu), ru(lenru)` are character, integer, and double precision arrays in which the user may store information between calls to `usrfun`.

3.2. The Initialization Routine

Subroutine `usrini` is used to initialize quantities associated with the problem. It is called once before Snopt.

```
      subroutine usrini( ObjAdd, ObjRow, Prob,
     &   x, xlow, xupp, xstate, Names,
     &   Fmul, Flow, Fupp, Fstate, FNames,
     &   iSpecs, iPrint, iSumm, iErr,
     &   cu,  iu,  ru,
     &   cw,  iw,  rw )
```

Each argument is fully described below. Many of the arguments are arrays (e.g., `x` is the vector containing an initial guess at the solution). However, we emphasize that *the lengths of the array arguments do not appear in the argument list*. The user must declare all arrays to be of fixed length at the head of the subroutine `usrini`. These declarations are used by the Perl script `snadiopt.pl` to automatically construct a main program that calls Snopt with appropriately dimensioned arrays. For example, a typical definition of the variables at the head of `usrini` is as follows.

```
      integer            n, neF
      integer            nName, nFnames
      integer            lencw, leniw, lenrw, lencu, leniu, lenru
      parameter        ( lencw = 501, leniw = 10000, lenrw = 20000 )
      parameter        ( lencu =   1, leniu =     1, lenru =     1 )
      parameter        ( n     =   5, neF   =     6                 )
      parameter        ( nName =   1, nFnames =   1                 )
      character*8        Names(nName), FNames(nFnames)
      double precision   x(n), xlow(n), xupp(n)
      double precision   Flow(neF), Fupp(neF), Fmul(neF)
      integer            xstate(n),  Fstate(neF)
      integer            iu(leniu), iw(leniw)
      double precision   ru(lenru), rw(lenrw)
      character*8        cu(lencu), cw(lencw)
```

The names of the arguments for `usrini` are unimportant, but the *position* of each argument is significant. For example, if the user prefers to call the vector of variables "`vars`" and declares the fourth argument of `usrini` to be

```
parameter          (maxvars = 5)
double precision    vars(maxvars)
```

then `snadiopt.pl` will parse this declaration and include

```
parameter          (n = 5)
double precision    x(n)
```

in the automatically generated calling routine. SnadiOpt can recognize Fortran style parameters and numbers but cannot read more complicated expressions. For instance, declaring `Fmul` as "`double precision Fmul(n+1)`" will definitely confuse it.

Below, we describe each of the arguments of `usrini`. In many cases, these arguments are assigned a default value in the automatically generated calling program. If the user wishes to use the default value of an argument, then it should not be altered in `usrini`. The symbol "∞" denotes the value of the Snopt optional parameter `Infinite bound`, which has default value 10^{20}.

Parameters:

`ObjAdd` is a double precision constant that is added to the objective function for printing purposes. `ObjAdd` does not affect the minimizer found.

Default value: `ObjAdd` $= 0.0$.

`ObjRow` is an integer defining the component of $F(x)$ to be used as the objective function $F_{obj}(x)$. If `ObjRow` $= 0$, then Snopt finds a point x that satisfies the constraints $l \leq x \leq u$, and $L \leq F(x) \leq U$,

Default value: `ObjRow` $= 1$.

`Prob` is an eight-character name for this model.

Default value: The name of the executable, truncated to eight characters.

`x` is a double precision array containing the point at which Snopt will start searching for a minimizer.

Default value: `x`$(j) = 0.0$.

`xlow, xupp` are double precision arrays containing the lower and upper bounds l and u such that $l \leq x \leq u$. By default, `xlow` and `xupp` are assumed to be infinite (i.e., the value of x is not restricted).

Default values: `xlow`$(j) = -\infty$, `xupp`$(j) = +\infty$.

`xstate` defines the initial state for each variable x. One may set `xstate`$(j) = 0$, `x`$(j) = 0.0$ for all $j = 1 : n$. All variables will be eligible for the initial basis.

Less trivially, to say that the optimal value of variable j will probably be equal to one of its bounds, set `xstate`$(j) = 4$ and `x`$(j) =$ `xlow`(j) or `xstate`$(j) = 5$ and `x`$(j) =$ `xupp`(j) as appropriate.

Default value: `xstate`$(j) = 0$.

`Names` is a character array of symbolic names for the components of `x`. Each name may have up to eight characters. If the user does not wish to supply symbolic names for the variables, `Names` should be declared to be be an array of length one.

`Fmul` is a double precision array of estimates of the dual variables for the constraints $L \leq F(x) \leq U$. (Dual variables are sometimes known as Lagrange multipliers or shadow prices.) `Fmul(ObjRow)` corresponds to the objective and is ignored.

Default value: `Fmul`$(j) = 0.0$.

`Flow, Fupp` are double precision arrays containing the lower and upper bounds L and U such that $L \leq F_i(x) \leq U$. The components `Flow(ObjRow)` and `Fupp(ObjRow)` corresponding to the objective is ignored. For an equality constraint of the form $F_i(x) = c$, set `Flow`$(j) =$ `Fupp`$(j) = c$.

Default values: `Flow`$(j) = -\infty$ and `Fupp`$(j) = +\infty$.

`FNames` is a character array of symbolic names for the constraints. Each name may consist of up to eight characters. If the user does not wish to supply names for the constraints, `FNames` should be declared to be an array of length one.

`iSpecs` is an open, readable Fortran file descriptor pointing to an options, or "specs" file. See the Snopt users guide to discover which options are available. If one chooses not to use an options file, `iSpecs` should be set to zero.

Default value `iSpecs` $= 0$.

`iPrint` is a Fortran file descriptor pointing to a file that will be overwritten with the results of this run of Snopt. If one does not wish to save the output to a file, `iPrint` should be set to zero.

Default value `iPrint` $= 0$.

`iSumm` a Fortran file descriptor pointing to a file that will be overwritten with summary information from this run of Snopt. Typically, `iSumm` is either set to 6, which will cause the summary output to be printed on the terminal, or set to 0, which disables the printing of summary information.

Default value `iSumm` $= 6$.

`iErr` a Fortran file descriptor pointing to a file that will be overwritten with diagnostic information from this run of Snopt. Set `iErr` to zero to disable printing of diagnostic information.

Default value `iErr` $= 0$.

`cu, iu, ru` are character, integer, and double precision arrays in which the user may store information between calls to `usrfun`.

`cw, iw, rw` are character, integer, and double precision work-space arrays used by Snopt. These arrays must be declared sufficiently large for Snopt to solve the optimization problem.

3.3. An Example Problem

Here we give examples of subroutines `usrini` and `usrfun` for the following four variable problem:

$$
\begin{array}{lrl}
\text{minimize} & 3x_1 + (x_1 + x_2 + x_3)^2 + 5x_4 & \\
\text{subject to} & 4x_2 + 2x_3 & \geq 0 \\
& x_1 + x_2^2 + x_3^2 & = 2 \\
& x_2^4 + x_3^4 + \quad x_4 & = 4 \\
& x_1 \geq 0 \qquad\quad x_4 & \geq 0.
\end{array}
$$

In the format of problem NP we have $L \leq F(x) \leq U$, where

$$L = \begin{pmatrix} -\infty \\ 0 \\ 2 \\ 4 \end{pmatrix}, \quad F = \begin{pmatrix} 3x_1 + (x_1 + x_2 + x_3)^2 + 5x_4 \\ 4x_2 + 2x_3 \\ x_1 + x_2^2 + x_3^2 \\ x_2^4 + x_3^4 + x_4 \end{pmatrix}, \quad U = \begin{pmatrix} +\infty \\ +\infty \\ 2 \\ 4 \end{pmatrix}.$$

The objective function has been assigned to the first component of F, which means that `ObjRow` $= 1$. The objective component is not constrained by Snopt, so there are infinite upper and lower bounds on F_{obj}. (A component with infinite upper and lower bounds is known as a "free row" of the problem.) Snopt automatically provides these infinite bounds on the objective row, and so it is unnecessary to provide them (unless later the user plans to set `ObjRow` $= 0$ to make Snopt find a feasible point for the constraints).

The upper and lower bounds on the variables are given by $l \leq x \leq u$, where

$$l = \begin{pmatrix} 0 \\ -\infty \\ -\infty \\ 0 \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}, \quad u = \begin{pmatrix} +\infty \\ +\infty \\ +\infty \\ +\infty \end{pmatrix}.$$

Our version of subroutine `usrini` performs four tasks: (i) it defines the length of the variable-dimensioned arrays used by Snopt and SnadiOpt; (ii) it opens the print file and summary file; (iii) it initializes the array of variables; and (iv) it defines the upper and lower bounds on x and F.

```
      subroutine usrini( ObjAdd, ObjRow, Prob,
     &     x, xlow, xupp, xstate, Names,
     &     Fmul, Flow, Fupp, Fstate, FNames,
     &     iSpecs, iPrint, iSumm, iErr,
     &     cu, iu, ru, cw, iw, rw )

      implicit           none
      integer            n, neF, nName, nFnames, ObjRow,
     &     lencw, leniw, lenrw, lencu, leniu, lenru
      parameter        ( lencw = 501, leniw = 10000, lenrw = 20000 )
      parameter        ( lencu =   1, leniu =     1, lenru =     1 )
      parameter        ( n     =   4, neF   =     4 )
      parameter        ( nName =   1, nFnames =   1 )
      integer            iSpecs, iPrint, iSumm, iErr, xstate(n),
     &     Fstate(neF), iu(leniu), iw(leniw)
      double precision   ObjAdd, x(n), xlow(n), xupp(n), Flow(neF),
     &     Fupp(neF), Fmul(neF), ru(lenru), rw(lenrw)
      character*8        Prob, Names(nName), FNames(nFnames),
     &      cu(lencu), cw(lencw)
*     ------------------------------------------------------------------
*     usrini defines input data for the problem discussed in the
*     SnadiOpt Users Guide.
*     ------------------------------------------------------------------
      integer            i
      character*20       lfile
      double precision   plInfy
      parameter         (plInfy = 1.0d+20 )
*     ------------------------------------------------------------------
*     Initial x.
```

```
      x(1)   =  1.0d+0
      x(2)   =  1.0d+0
      x(3)   =  1.0d+0
      x(4)   =  1.0d+0

      xlow(1) = 0.0d+0
      xlow(4) = 0.0d+0

*     Impose bounds on the constraint rows.

      Flow(2)   =  0.0d+0
      Flow(3)   =  2.0d+0       ! Equality constraint
      Fupp(3)   =  2.0d+0
      Flow(4)   =  4.0d+0       ! Equality constraint
      Fupp(4)   =  4.0d+0

      iSpecs = 4
      iPrint = 15

      lfile = 'prob.spc'
      open( iSpecs, file=lfile, status='OLD',     err=800 )
      lfile = 'prob.out'
      open( iPrint, file=lfile, status='UNKNOWN', err=800 )

      return

  800 write(iErr, 4000) 'Error while opening file', lfile
 4000 format(/  a, 2x, a  )

      end ! subroutine usrini
```

Note that default initial values are used for the variables `Prob`, `Fmul`, `xstate`, `Fstate`, and `ObjAdd`. Similarly, only those bounds not equal to their default infinite values are assigned.

The subroutine `usrfun` defines the values of the vector $F(x)$.

```
      subroutine usrfun( Status, mode,
     &     neF, n, x, F,
     &     cu, lencu, iu, leniu, ru, lenru,
     &     cw, lencw, iw, leniw, rw, lenrw )

      implicit           none
      integer            Status, mode, neF, n, lencu, leniu, lenru,
     &     lencw, leniw, lenrw, iu(leniu), iw(leniw)
      double precision   F(neF), x(n), ru(lenru), rw(lenrw)
      character*8        cu(lencu), cw(lencw)
*     ------------------------------------------------------------------
*     Usrfun computes the objective and constraint functions for the
*     problem featured in the SnadiOpt users guide.
*     ------------------------------------------------------------------
      integer            Obj
*     ------------------------------------------------------------------
      Obj  = 1                  ! The objective row

      F(Obj) = 3.0d+0*x(1) + (x(1) + x(2) + x(3))**2 + 5.0d+0*x(4)
```

```
*     Constraint functions.

      F(2)   =              4.0d+0*x(2)    + 2.0d+0*x(3)
      F(3)   =       x(1) +        x(2)**2 +        x(3)**2
      F(4)   =                     x(2)**4 +        x(3)**4  + x(4)

      end ! subroutine usrfun
```

4. Invoking SnadiOpt

The user-supplied routines must be run through ADIFOR and then compiled and linked into a complete program, before Snopt may be invoked. There are two steps in the process of building this complete program. First, the user invokes the script `snadiopt.pl` to scan the user-supplied routines and produce the components that are needed to build a complete executable, notably the main program itself. Second, the user invokes a version of the program `make` to perform the build.

Users will typically call `snadiopt.pl` only once. Most changes made to a model can be incorporated into the executable by simply typing `make`. By design, `snadiopt.pl` generates a (relatively) straightforward set of components that may be modified at will. In some cases, particularly if the sizes of the arrays defined in `usrini` change, it may be convenient to call `snadiopt.pl` again rather than modifying multiple files. We have incorporated a feature into `snadiopt.pl` to simplify this process. See Section 4.4 for more information.

4.1. Locating Executables and Libraries

Before one can use SnadiOpt, the package must, of course, be installed on the user's machine. Installation instructions are provided with the SnadiOpt distribution. Because the installation process depends on the machine type and the source of the distribution, installation instructions will not be repeated here. A few words, however, are in order.

The SnadiOpt package consists of a program named `snadiopt.pl`, some data files needed by `snadiopt.pl`, and code libraries that must be combined (linked) with user-supplied code to produce a problem executable. Ideally, these components will be installed in an appropriate, system-dependent location. On Unix systems, for instance, the default location is the `/usr/local/` directory structure. If the SnadiOpt package is located in some appropriate system location, then typing

```
% snadiopt.pl --help
```

will provide summary help information about using the `snadiopt.pl` program. If this is the case, then the rest of this section may be skipped. If the system cannot find the `snadiopt.pl` program, then the user will need to tell the system where to find it. The following instructions are for versions of the Unix operating system.

First, one should ask the individual who installed SnadiOpt where the `snadiopt.pl` program is located. If it is not in a system-dependent location, it will normally be found in the `bin` subdirectory of the Snopt distribution. The `cd` command may be used to change the working directory to the directory containing `snadiopt.pl`. The command used to set the `PATH` environment variable depends on which shell is being used. For `bash` or `ksh`, the appropriate command is

```
% PATH=$PATH:$PWD
```

but for `csh` or `tcsh`, the command

```
% setenv PATH ${PATH}:${PWD}
```

must be used. Virtually all users will be using a shell that responds to one of these two commands. It is safe to try these commands if one is unsure which shell is being used.

Once the `PATH` environment variable has been set, the system will be able to find `snadiopt.pl`. One can then generate and compile problem executables. The executables may not, however, run. While this would seem to be undesirable behavior, there is actually a reason for it. Modern operating systems support the concept of dynamically linked libraries. Such libraries are not copied into an executable, but rather are loaded into memory when a program is run. With such a scheme, several executables may share one library. SnadiOpt uses dynamically linked libraries whenever possible.

Because dynamically linked libraries must be loaded at run-time, the operating system must know where to locate these libraries. The simplest scheme is to place the libraries in a system-dependent location. If one had to set the `PATH` environment variable to tell the system where to find the `snadiopt.pl` program, then it is likely that one will need to tell the system where to find the SnadiOpt libraries as well. To do so, one should first change the working directory to the directory containing the SnadiOpt libraries. These libraries will usually be located in the `lib` subdirectory of the Snopt distribution, and will have names similar to `libsnddiopt.so` and `libsnsdiopt.so`. If the user is using `bash` or `ksh`, the command

```
% LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD
% export LD_LIBRARY_PATH
```

should be executed. Those using `csh` or `tcsh`, must determine whether `LD_LIBRARY_PATH` has already been set. If the command

```
% printenv LD_LIBRARY_PATH
```

prints nothing, then the variable has not been set, and

```
% setenv LD_LIBRARY_PATH ${PWD}
```

will set it appropriately. Otherwise, typing

```
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${PWD}
```

will add the current directory to the existing `LD_LIBRARY_PATH`.

Let us make a few, final notes about this process. These days, it is common for a user to have multiple command windows open. It is a frequent mistake to think that setting the `PATH` variable, or any variable, in one window sets its value in all windows. Setting a variable in one command window does not affect the other command windows in any way. It is usually possible to alter certain initialization files to set the values of `PATH` and `LD_LIBRARY_PATH` in every command window automatically at login. It is not possible for us to describe this process, because it is very system dependent. One should ask a system administrator how to do this.

4.2. Basic Usage

Suppose a user has placed all the code needed to define a certain problem, including the required subroutines `usrfun` and `usrini`, in a file named `prob.f`. The command

```
% snadiopt.pl -o prob prob.f
```

will generate the files that are needed to build an executable named `prob` that solves the user's optimization model.

To actually build the executable, invoke the GNU version of the program `make`. On many systems, GNU make is installed as `make` or `gmake`, and so typing

```
% make
```

or

```
% gmake
```

should build a program named `prob` that may be executed from the command line.

4.3. Files Generated by `snadiopt.pl`

In this section, we briefly describe the files generated by `snadiopt.pl` itself. Other temporary files may be generated by the compiler and ADIFOR. A beginning user should not need to know about the generated files in order to use this package. Therefore, this section may be skipped on a first reading.

This section lists the files generated for a problem named `prob`. The script `snadiopt.pl` uses "prob" as the prefix for most of the files generated for this problem. If the user had invoked the command

```
% snadiopt.pl -o prob prob.f
```

then "prob" will be the prefix of the generated files. In general, the `-o` option determines the prefix of the generated files. If the option is omitted, then "unnamed" will be the prefix used.

`GNUmakefile`. This is the only generated file that is not prefixed by the name of the problem. The `GNUmakefile` is meant to be shared by all problems in a given directory; it contains general information about building and managing problem executables.

Users may wish to modify `GNUmakefile` to customize the build process. For instance, `GNUmakefile` might be modified to tell the compiler to generate object code suitable for use with a debugger. The `snadiopt.pl` program will not overwrite an existing `GNUmakefile`, unless it is called with the `--refresh-makefile` option. Therefore, it is safe to modify this file.

`prob_submake`, `prob_submake.orig`, `prob_submake.bak`. The file `prob_submake` contains the commands for building the program, including the commands for calling ADIFOR . The file should contain the complete dependency information for the program and should be capable of rebuilding the program when components are modified.

It is sometimes necessary to modify `prob_submake`. The file `prob_submake.orig` contains the original version of this file, as generated by `snadiopt.pl`. This allows the user to compare the modified version of `prob_submake` with the original file.

If `snadiopt.pl` detects an existing file named `prob_submake`, it will save this file as `prob_submake.bak`. The user may then reapply any changes made to `prob_submake` to the newly generated file. Many times, these changes can be merged automatically. See Section 4.4 for more information.

`prob_main.f`, `prob_main.f.orig`, `prob_main.f.bak`. The file `prob_main.f` contains the Fortran main program that calls Snopt with the user's data and problem definition functions. It also performs some necessary bookkeeping and initialization and is responsible for allocating the arrays that the user requests in the `usrini` subroutine.

Users may wish to modify `prob_main.f`. For instance, a user might wish to output the results from Snopt in a particular format and so might place the commands for doing so in `prob_main.f`. The file `prob_main.f.orig` contains the version of `prob_main.f` generated

by the last call to `snadiopt.pl`. This allows the user to compare the modified `prob_main.f` with the original file.

If `snadiopt.pl` detects an existing file named `prob_main.f`, it will save that file as `prob_main.f.bak` before proceeding. All changes that the user had made to the existing `prob_main.f` are saved in that back-up file, and the user may reapply these changes to the newly generated file. Many times, these changes can be merged automatically. See Section 4.4 for more information.

`prob.adf`. The file `prob.adf` contains the ADIFOR "script" for differentiating the model's functions. See the ADIFOR users guide for more information. It is unlikely that a user will need to modify this file.

`prob_admain.f`. ADIFOR requires a complete compilable program in order to differentiate a function called from that program. The file `prob_admain.f` contains a phony program that calls `usrfun`. We know of no reason for users to modify this file.

`prob_sparse_dispatch.f`, `prob_dense_dispatch.f`. These files call library routines supporting the use of ADIFOR with Snopt. The existence of these files is an artifact of the Fortran language not having syntax for storing a reference to a subroutine. We know of no reason for users to modify these files.

`prob.cmp`. The file `user.cmp` is not generated by `snadiopt.pl`, but rather is created as an intermediate file in the build process. It contains a list of Fortran files that are to be sent to ADIFOR. One should not not modify this file; one should modify the `AD_SOURCE` and `AD_OTHER_FILES` variables in the file `prob_submake`.

4.4. Merging Changes

Users need to call `snadiopt.pl` only once. The components that it creates may then be modified at will, and the executable rebuilt using `make`. However, on some occasions it may be useful to call `snadiopt.pl` again, particularly when

- the sizes of the arrays in `usrini` have changed. The array sizes in `prob_main.f` and possibly `prob.adf` must also be modified. The program `snadiopt.pl` will update these quantities automatically.
- the names of the parameters of `usrfun` have changed. The user must either call `snadiopt.pl` again, or edit `prob.adf` to update the names of the independent variables (`AD_IVARS`) and the names of the dependent variables (`AD_DVARS`).
- the names of Fortran source files are modified, or new source files are added. Users will need to update `prob_submake`, or call `snadiopt.pl` again.

It is not uncommon, however, for users to want to modify `prob_submake` to customize the build process, or modify `prob_main.f` to perform some action on the results of `snopt`. Normally, when `snadiopt.pl` is called, it overwrites these files, saving copies of the existing files as `prob_main.f.bak` and `prob_submake.bak`. Users may then reapply the changes they had made to the old `prob_main.f` and `prob_submake` to the newly generated files.

There are, however, Unix utilities that are able to merge changes between versions automatically. The `snadiopt.pl --merge` option provides an interface to these tools. Simply call `snadiopt.pl` with the arguments

```
% snadiopt.pl --merge -o prob prob.f
```

The merge is based only on the comparison of blocks of text. It does not pretend to understand the meaning of the code. However, it is effective remarkably often. In case the merge is ineffective, the files that `snadiopt.pl` would have produced without the merge option may be found in `prob_submake.orig` and `prob_main.f.orig`.

Rarely, there will be conflicts that make it impossible to complete the merge. In these cases, lines of the form

```
<<<<<<< prob_submake.bak
lines from prob_submake.bak
=======
lines from prob_submake.orig
>>>>>>> prob_submake.orig
```

will be inserted in the files, and these sections must be edited by hand.

The merge option uses the standard Unix utilities `diff3` and `ed`. Merging is not supported on platforms on which these programs are not available. We don't support automatic merging of the other generated files. Merging requires that the files `prob_main.f.orig` and `prob_submake.orig` generated by the last call to `snadiopt.pl` be present in the current directory.

4.5. Advanced Usage

Multiple Source Files. The script `snadiopt.pl` is not restricted to scanning a single file. If several Fortran files are needed to define the problem, all file names should be included on the command line.

Library Source Files. ADIFOR understands Fortran intrinsics, operators such as `sqrt` that look like functions but actually have special status in the language. It must, however, have the source code to actual functions used in the program, such as the functions defined in the BLAS [8]. Source files for these functions must be included on the command line.

If a user is certain that the included library functions do not need to be differentiated, and would rather link against the installed library than recompile, he may include the source file names in the `AD_OTHER_FILES` variable in the `prob_submake`. See Section 5 for more information.

Using Alternative Function Names The `snadiopt.pl` script tries to be flexible about the names of the problem definition functions. Several options that allow these names and the names of certain output files to be specified by the user. The available options are summarized in Section 4.6.

4.6. Summary of All Options

```
Usage: snadiopt.pl [switches] file1.f [file2.f]
   -help                 Print this message.
   -version              Print the version number of snadiopt.pl.
   -o PROGRAM            The optimization problem (and binary executable)
                         will be named PROGRAM. (default: a.out)
   -makefile MAKEFILE    The output makefile will be named MAKEFILE.
                         (default: PROGRAM_submake or unnamed_submake
                         if PROGRAM is not specified.)
   -refresh-makefile     Create MAKEFILE, even if it already exists.
                         Unless given this option, the script will not
```

```
                    overwrite an existing MAKEFILE.
-usrfun NAME        The FORTRAN subroutine named NAME computes the
                    functions needed in this optimization problem.
                    (default: usrfun)
-usrini NAME        The FORTRAN subroutine named NAME initializes
                    this optimization problem. (default: usrini)
-merge              Merge changes between prob_main.f.orig and
                    the current version of prob_main.f into the newly
                    generated prob_main.f. Do the same for
                    prob_submake.
```

5. Building the Executable

The Unix Make utility is used to generate targets, in this case executables that solve specific optimization problems, from source files. The rules that Make uses to build these targets are specified in files known as makefiles. The Make utility is also commonly used to perform certain bookkeeping tasks, such as removing files generated by the build process.

This project uses the GNU dialect of Make. This dialect has certain pattern substitution features that are absent in other versions of Make. Furthermore, GNU Make is freely available on virtually every platform. Vendor-specific versions of Make are not consistent in interface, language, or quality. Thus, we use GNU make to get predictable performance on a wide variety of platforms.

We assume that the reader has a basic knowledge of the Make utility. (For a good introduction to Make, see [10] or [11].) This section describes how we have arranged our makefiles, the targets that are available, and variables that may be modified to effect the build process. Most users will simply invoke GNU Make without any arguments to build all problems in the current directory (provided that `snadiopt.pl` has already been invoked to create the necessary components.) For the rest of this section, we assume that GNU Make has been installed and may be invoked by the command `make`. Users should substitute the command that they use to invoke GNU Make wherever appropriate.

On Unix systems, if the SnadiOpt package is not installed in an appropriate system location, the user may have to set the `LD_LIBRARY_PATH` environment variable before the executables that are built will run. See Section 4.1 for instructions on how to do this.

5.1. Typical Usage

Before invoking Make, users must call `snadiopt.pl` to generate the components of each problem they wish to build. Then, typing `make` will cause executables to be built for all models in the current directory. If users wish to build executables for only some of the problems, they may instead list the names of the executables that they wish to build. For instance;

```
% make prob1 prob3
```

would build only the executables `prob1` and `prob3`.

5.2. Subordinate Makefiles

Traditionally, Make takes all its input from a single file, typically named `makefile`, `Makefile`, or `GNUmakefile`. This scheme has proven to be too restrictive in practice, so many versions of Make, and GNU make in particular, support the `include` directive. A line of the form

```
include filename
```

tells Make to act is if all the text in "filename" were included literally in the makefile.

In a problem directory, there will be a single file named `GNUmakefile` and one or more files with the suffix "submake." Each of the files with an appended `_submake` is called a subordinate makefile, because it does not contain a complete set of rules and dependencies for building the executable for a problem. The `GNUmakefile` uses the `include` directive to include the text of all the subordinate makefiles. Each model in a directory will have its own subordinate makefile, which will contain the specific rules, variables, and dependencies for building an executable that optimizes that model. The text of `GNUmakefile` contains generic rules and dependencies that are needed to build any model.

Subordinate makefiles are useful for several reasons:

- Users may wish to have more than one model in a directory. Having complete, separately named makefiles for each model becomes awkward, requiring the user to specify the name of the makefile for each build.

- Users may want to build the executable for more than one model or to take some other action that affects more than one model. When subordinate makefiles are used, the rules for all the models are available, so a user may build any combination of targets by typing their names on the command line. The command "`make all`" works as expected and is the default target.

- Sometimes multiple models will share one or more files. Because Make is given the complete set of dependencies for the executables of all the models, it can quickly determine which files need to be rebuilt. If each model had a separate makefile, the user would have to make this determination or, alternatively, call Make once for every executable.

The Make program can scan all the subordinate makefiles and build a complete set of dependencies quickly. The time taken is typically many times shorter than the time needed to compile a single file. Some users may be surprised by this speed. See [9] for a discussion of issues affecting the efficiency of Make.

5.3. Useful `makefile` Targets

In addition to the names of the programs, a number of "phony" targets may be specified for Make. These targets cause some action to be taken, rather than causing the target to be built. These targets commonly are defined to perform useful project-management tasks, such as deleting "`.o`" files.

`all` builds everything. This is the default, so `make all` is equivalent to Make.

`check` checks the consistency of the Fortran files used to build each program. This requires that `ftnchek` has been installed. The `ftnchek` program and documentation are freely available and may be obtained from `http://www.netlib.org`.

`clean` removes object ("`.o`") files and some common "garbage" files, such as `core` files. This does not remove the executable file or any files generated by ADIFOR.

`veryclean` removes more files generated by the build process, including the executable and all output from ADIFOR. This target also removes files named "`prob.out`", the traditional name for output from the solvers.

`distclean` cleans up for distribution. This target is like `veryclean` but does not delete the differentiated Fortran problem files, since those files are considered part of a distribution. Use this target to distribute the files to a machine on which the ADIFOR translator is not available. This target invokes `clean`, `adifor-clean` and `snadiopt-clean`.

`maintainer-clean` deletes everything that can be rebuilt. This includes the files created by a call to `snadiopt.pl` and the makefiles themselves.

`adifor-clean` removes ADIFOR auxiliary files, but not the autodifferentiated Fortran files.

`adifor-veryclean` removes ADIFOR auxiliary files and the differentiated Fortran files (and `*.cmp`).

`snadiopt-clean` removes auxiliary files generated by SnadiOpt for use with all the programs. These are the "`.bak`" and "`.orig`" files.

`snadiopt-veryclean` removes all files generated by SnadiOpt for use with all the programs. Files generated by SnadiOpt cannot be rebuilt by using commands in the makefile. This target is intended to reverse the effect of calling `snadiopt.pl`.

These targets also have versions that are limited to the components of a single module. For instance, `make prob-clean` will remove auxiliary files generated in a build of the executable `prob`. In general, any of these targets may be prefixed by the name of a specific problem.

5.4. Useful `makefile` Variables

Each subordinate makefile contains the following variables that users might need to modify.

`prob_USER_LIBS` defines any libraries that need to be linked with the user's code to produce an executable. The SnadiOpt libraries are automatically included.

`prob_SOURCE` is a list of all Fortran source files for the model `prob`. A reference to the variables `prob_AD_SOURCE` and `prob_AD_G_SOURCE` should appear in this list.

`prob_AD_SOURCE` is the list of files to be differentiated by ADIFOR.

`prob_AD_OTHER_FILES` is a list of files that must be passed to ADIFOR in addition to those of `prob_AD_SOURCE` in order to make a complete program. The files in this list differ from the files in `prob_AD_SOURCE` in that neither the original file nor the result computed by ADIFOR need be compiled into the problem executable. Phony main programs and phony library stubs belong on this list.

5.5. Dense ADIFOR

Snopt is a sparse optimization solver; its internal data is stored in sparse matrix format. Sparse matrix format is designed to take advantage of the fact that many the elements of the matrix will be identically zero.

ADIFOR can either generate derivative code that uses sparse matrices internally or code that uses dense matrices internally. For small problems, typically problems with fewer than 30 variables, the derivative code generated by dense ADIFOR can be more efficient. This is typically not an important issue unless the problem is highly nonlinear (otherwise, it is just a small simple problem and will be solved quickly regardless of which version of ADIFOR is used.)

A sequence of commands of the form

```
% make adifor-veryclean
% make AD_FLAVOR=dense
```

will cause ADIFOR to generate code that uses dense matrices to compute derivatives. It is important to `make adifor-veryclean` whenever switching between dense and sparse versions of ADIFOR. The `make` program is unable to tell that a user has switched versions of ADIFOR, and it thus cannot tell which files need to be rebuilt.

References

[1] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *ADIFOR: Generating derivative codes from Fortran programs*, Sci. Programming, 1 (1992), pp. 11–29.

[2] C. Bischof, A. Carle, P. Khademi, and A. Mauer, *ADIFOR 2.0: Automatic differentiation of Fortran 77 programs*, IEEE Computational Science & Engineering, 3 (1996), pp. 18–32.

[3] A. Brooke, D. Kendrick, and A. Meeraus, *GAMS: A User's Guide, Release 2.25*, Scientific Press, 1988.

[4] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press/Brooks-Cole Publishing Company, 1993.

[5] P. E. Gill, W. Murray, and M. A. Saunders, *SNOPT: An SQP algorithm for large-scale constrained optimization*, Numerical Analysis Report 97-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997.

[6] ———, *User's guide for SNOPT 5.3: a Fortran package for large-scale nonlinear programming*, Numerical Analysis Report 97-5, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997.

[7] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, London, 1981.

[8] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for FORTRAN usage*, ACM Trans. Math. Soft., 5 (1979), pp. 308–323.

[9] P. Miller, *Recursive make considered harmful.* http://www.pcug.org.au/m̃illerp/rmch/recu-make-cons-harm.html.

[10] A. Oram and S. Talbott, *Managing Projects with make, 2nd edition*, O'Reilly & Associates, 1991.

[11] R. M. Stallman and R. McGrath, *GNU Make, A Program for Directing Recompilation*, Free Software Foundation, April 2000.

Index